**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Enis Novak**

# POTPROGRAMI U PROGRAMSKIM JEZICIMA

**ZAVRŠNI RAD**

**Varaždin, 2018.**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Enis Novak**

**Matični broj: 41960/13–R**

**Studij: Informacijski sustavi**

# POTPROGRAMI U PROGRAMSKIM JEZICIMA

**ZAVRŠNI RAD**

**Mentor/Mentorica:**

Prof. dr. sc. Alen Lovrenčić

**Varaždin, 2018.**

*Enis Novak*

**Izjava o izvornosti**

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

*Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi*

_____

**Sažetak**

Programski jezici su kompleksni alati koji se koriste za pisanje programa, a kako bi se olakšalo njihovo korištenje i dodale nove mogućnosti koriste se potprogrami. Potprogrami su blokovi naredbi koji izvršavaju jedan zadatak neke veće cjeline. Izvrsni i spretni potprogrami uvelike smanjuju trošak razvoja i održavanja velikog programa. Potprogram se u različitim razvojnim alatima još naziva funkcija, metoda ili rutina. Za sam poziv potprograma se koristi struktura podataka zvana „kontrolni stog“ koji se sastoji od pokazivača stoga, okvira potprograma, varijabli te povratne adrese. Potprogram ujedno može biti i rekurzija, odnosno potprogram koji poziva samog sebe. Postoje tehnike zvane „preopterećivanje“ i „nadjačavanje“ koje daju dodatnu dubinu i mogućnosti u korištenju potprograma. Svaki potprogram se mora pozvati kako bi se izvršio što se može učiniti direktnim ili indirektnim pozivom.

**Ključne riječi:** potprogram, programski jezik, stog, funkcija, rekurzija, poziv potprograma, inline.

# Sadržaj

# 1. Uvod

Svaki programski jezik omogućuje pisanje bloka koda koji, kada je pozvan, obavlja određeni zadatak. U programiranju ti blokovi koda se nazivaju potprogrami. Potprogram je redoslijed programskih uputa koje obavljaju određeni zadatak kao jedinica.

Sam naziv *potprogram* sugerira da se ponaša na isti način kao i računalni program koji se koristi kao dio većeg programa ili drugog potprograma. Potprogrami su često kodirani na način koji omogućuje pozivanje nekoliko puta i to sa više različitih mjesta, uključujući i druge potprograme, te se nakon obavljenog zadatka vrate na sljedeću uputu nakon poziva.

Potprogrami razbijaju veće računalne zadatke u manje te omogućuju ljudima da se nadovezuju na tome što su drugi već napravili umjesto da počinju ispočetka. Prikladni potprogrami se ponašaju kao crna kutija, sakrivaju detalje o svojim funkcionalnostima koje nije potrebno znati kako bi se lakše koristile, te sve što programer treba znati o određenom potprogramu su njegovi ulazi i izlaz.

Potprogrami mogu biti definirani unutar samog programa ili zasebno u bibliotekama koje se onda mogu koristiti u više različitih programa. U različitim programskim jezicima potprogrami se mogu nazivati procedura, funkcija, rutina ili metoda.

Potprogrami se koriste za podjelu različitih funkcionalnosti programa te se često kombiniraju da odrade kompliciraniji zadatak. Također se koriste za uklanjanje repetativnosti čime se smanjuje broj linija koda te ga je lakše i jednostavnije održavati. Potprogrami obično imaju zajedničku strukturu bez obzira na programski jezik koji se koristi.

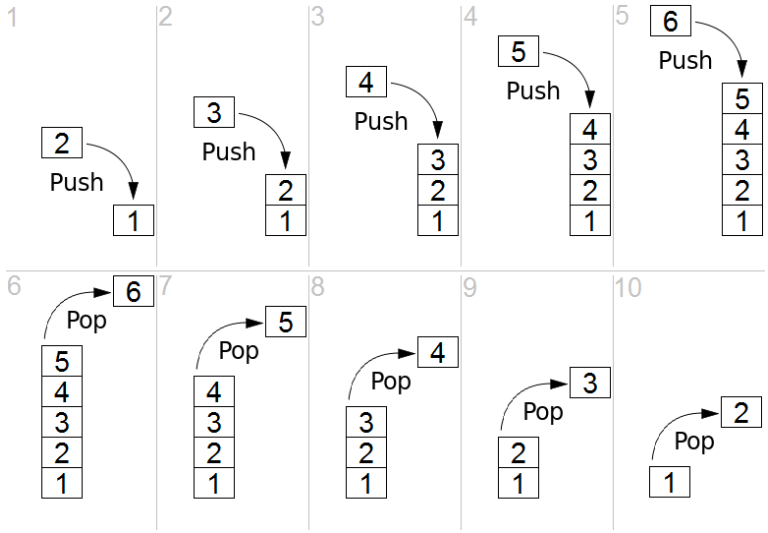
```
tip ime (parametri){
      tijelo potprograma
}
```

Potprogrami se mogu kodirati tako da se rekurzivno pozivaju na jednom ili više mjesta što omogućava izravnu implementaciju funkcija definiranih matematičkim indukcijama. Potprogram može vratiti izračunatu vrijednost svom pozivatelju (povratna vrijednost) ili dati različite vrijednosti rezultata ili izlazne parametre. Uobičajena svrha potprogramske jedinice je samo izračunati jedan ili više rezultata čije vrijednosti su u potpunosti određene argumentima koji se prenose u potprogram.

# 2. Stog (stack)

Stog (eng. *Stack*) je dinamička struktura podataka koja podržava operacije dodavanja i brisanja elementa. Ti elementi se isključivo dodavaju ili oduzimaju sa stoga. Svaki stog ima osnovne operacije *Push* pomoću koje se elementa dodaje na stog i *Pop* pomoću koje se element briše sa stoga. Postoji više principa po kojem se elementi mogu dodavati i oduzimati poput First-In-First-Out (FIFO) ili kružnog stoga no najčešće korišteni je Last-In-First-Out (LIFO). Stog koji se koristi za praćenje potprograma se zove „pozivni stog“ ili „kontrolni stog“ (eng. *Call stack*) te se često skraćuje na „stog“.

Primjer stoga:




Slika 1. Primjer dodavanja i brisanja elementa sa stoga pomoću Push i Pop operacija (preuzeto sa https://en.wikipedia.org/wiki/Stack_(abstract_data_type) )

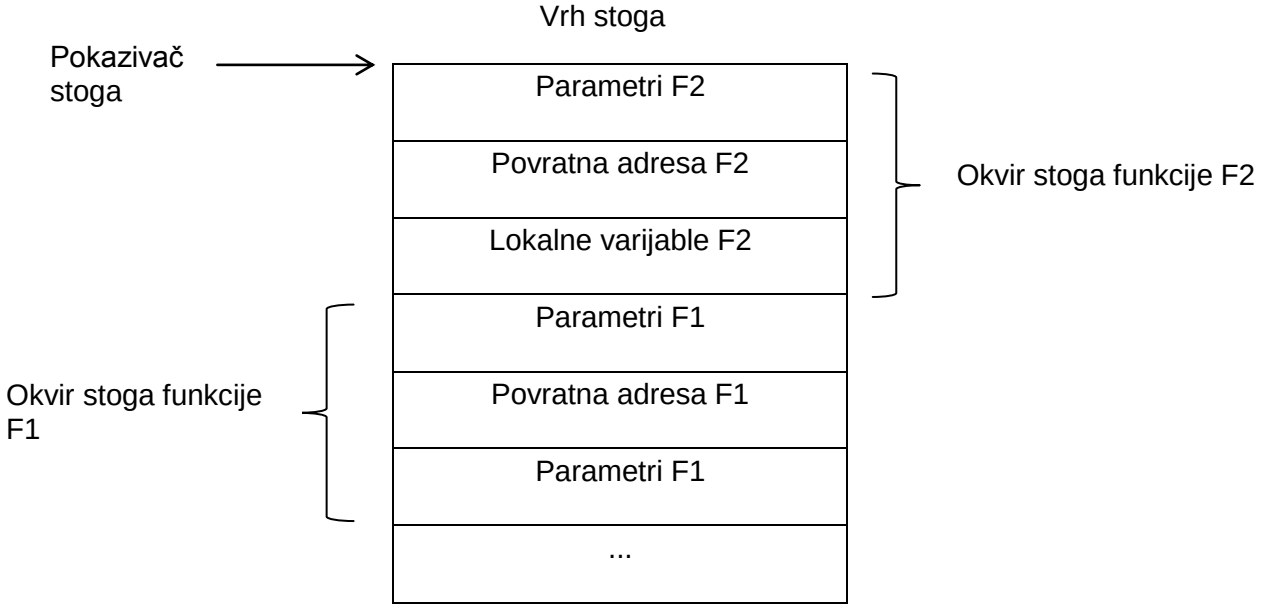
Upravljanje tim stogom je često sakriveno u programskim jezicima više razine. U arhitekturi računala stog je struktura podataka koja omogućava spremanje informacija o aktivnim potprogramima, odnosno funkcijama, unutar računalnog programa. Sve funkcije se prilikom izvršavanja reduciraju na strojne instrukcije koje procesor učitava iz memorije tako da zna adrese tih instrukcija.

Postoji nekoliko važnih primjena stogova od kojih je najvažnija u implementaciji poziva funkcija. Primarna svrha stoga je praćenje točke povratka (eng. *return point*) u kojem aktivne funkcije vraćaju kontrolu nakon svog završetka. Aktivna funkcija je ona koja je pozvana te se mjesto (adresa) instrukcije na kojem se pozvana funkcija kasnije može vratiti mora biti negdje spremljena. Tipični kontrolni stog se sastoji od povrate adrese (eng. *return address*), lokalnih varijabli i parametara.

## 2.1. Struktura kontrolnog stoga

Kontrolni stog se sastoji od okvira. Svaki okvir odgovara pozivu potprograma koja još nije terminiran s povratom (return). Okvir na vrhu stoga odgovara trenutno izvršavanom potprogramu i sadrži sljedeće informacije

- ukoliko postoje argumenti (parametri) koji se prosljeđuju potprogramu - potprogrami koji su pozvani često imaju vrijednosti koje im predaju potprogrami koji ih pozivaju pa ih je poželjno imati u kontrolnom stogu
- povratna adresa - kada je unutar nekog programa ili potprograma (F1) pozvan drugi potprogram (F2), F2 treba znati lokaciju (adresu) instrukcije na kojoj dalje može nastaviti potprogram F1 kada on terminira
- lokalne varijable potprograma - potprogram treba memorijski prostor za spremanje lokalnih varijabli koje su dostupne samo tom potprogramu i sama vrijednost tih varijabli je potrebna u vremenu aktivnosti tog potprograma. Stoga se memorijski prostor alocira unutar stoga za lokalne varijable



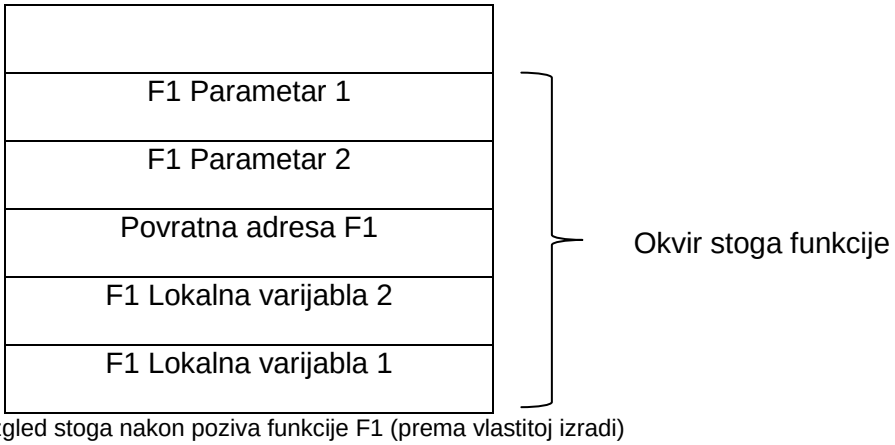

Tablica 1. Opći primjer kontrolnog stoga (prema vlastitoj izradi)

U sljedećem primjer C++ koda je objašnjeno što se to točno događa u kontrolno stogu prilikom poziva funkcija.

```
#include <iostream>
using namespace std;

void F1(int p1, int p2);
void F2(int p1, int p2);
```

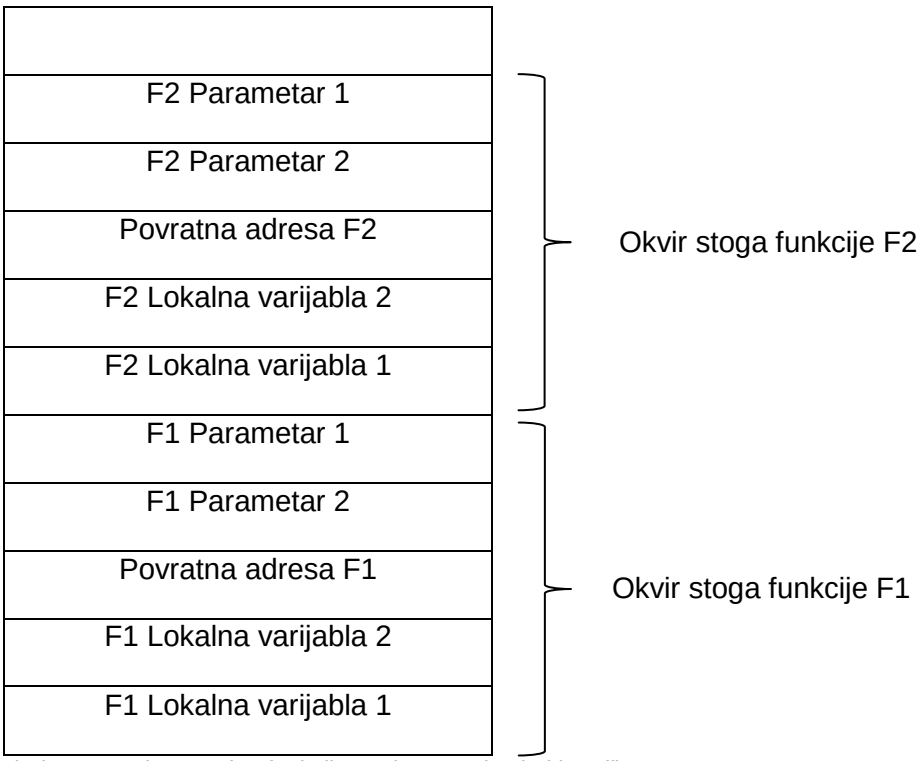
```
void F3(int p1, int p2);

void F1(int parametar1, int parametar2){
	int lokalna1 = 0; int lokalna2 = 0;
	cout << "Funkcija 1 poziva funkciju 2." << endl;
	F2(lokalna1, lokalna2);
}

void F2(int parametar1, int parametar2){
	int lokalna1 = 0; int lokalna2 = 0;
	cout << "Funkcija 2 poziva funkciju 3." << endl;
	F3(lokalna1, lokalna2);
}

void F3(int parametar1, int parametar2){
	int lokalna1 = 0; int lokalna2 = 0;
	cout << "Funkcija 3 ne radi ništa korisno." << endl;
}

int main(void){
	int lokalna1 = 0; int lokalna2 = 0;
	F1(lokalna1, lokalna2);
	return 0;
}
```

Program se sastoji od tri funkcije te glavnog bloka (main). U *main* bloku imamo dvije varijable i poziv funkcije F1. Kontrolni stog je prije poziva funkcije F1 prazan. Nakon što se pozove funkcija F1, na stog se spremaju njene lokalne varijable, povratna adresa te parametri.



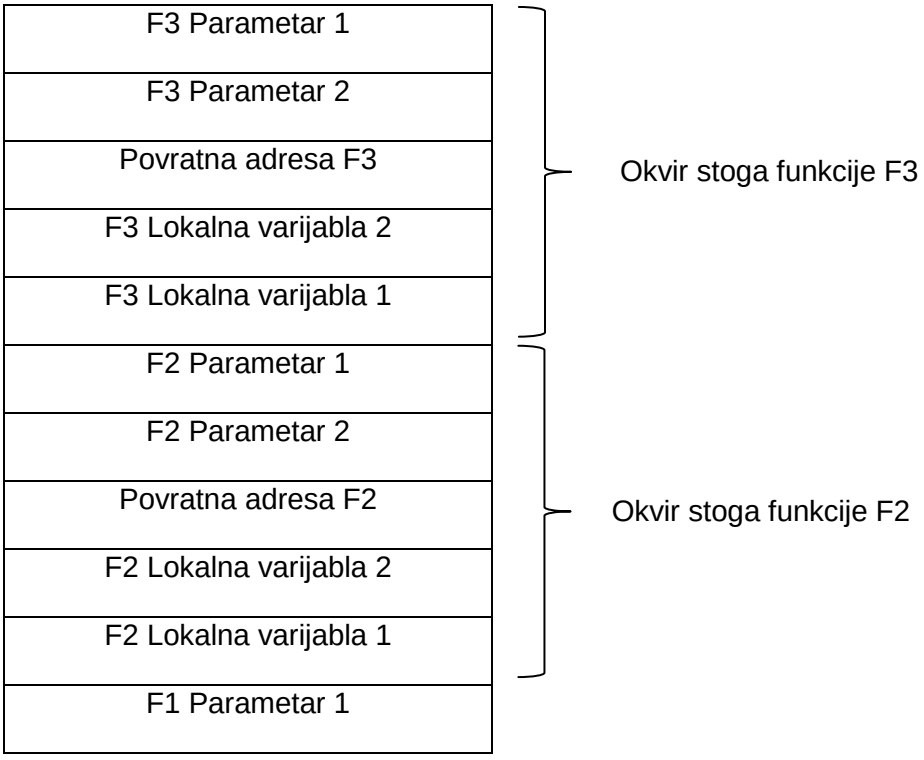

Tablica 2. Izgled stoga nakon poziva funkcije F1 (prema vlastitoj izradi)

Pošto funkcija F1 poziva funkciju F2 prije nego je F1 završio sa radom, na stog se dodaju parametri funkcije F2.




Tablica 3. Izgled stoga nakon poziva funkcije F2 (prema vlastitoj izradi)

Pošto funkcija F2 poziva funkciju F3 prije nego je F2 završio sa radom, na stog se dodaju parametri funkcije F3.



| |
|---|
| F1 Parametar 2 |
| Povratna adresa F1 |
| F1 Lokalna varijabla 2 |
| F1 Lokalna varijabla 1 |

Tablica 4. Izgled stoga nakon poziva funkcije F3 (prema vlastitoj izradi)

Nakon što funkcija F3 završi sa radom, vraća kontrolu natrag na funkciju F2 te se pomoću povrate adresa sa stoga vraća na prethodnu instrukciju te nastavlja sa radom.

# 3. Rekurzije kao funkcijski pozivi

Postoje funkcije i procedure koje pozivaju same sebe kako bi obavile neko računanje. Takve funkcije/procedure se zovu rekurzije. Uz pomoć kontrolnog stoga omogućeni su višestruko ugniježđeni funkcijski pozivi i kao poseban slučaj su upravo rekurzije.

Dijelovi rekurzivnog algoritma

1. Bazni slučaj – slučaj u kojem rekurzija završava
2. Koraci prema baznom slučaju – dio u kojem se problem pojednostavnjuje, odnosno gdje se isti algoritam koristi kako bi se rješio pojednostavnjeni problem
3. Rekurzivni poziv – funkcija poziva samu sebe

Rekurzije se ponašaju kao i sve ostale funkcije te koriste stog za spremanje lokalnih vrijednosti, parametara i povratne adrese kao što je objašnjeno u prethodnom primjeru.

Primjer rekurzije:

```
#include <iostream>
using namespace std;

int fibonacci(int x) {
    if (x == 0) return 0;
    if (x == 1) return 1;

    return fibonacci(x - 1) + fibonacci(x - 2);
}

int main(void){
      int fibo;
      cout << "Unesi koji broj u Fibonnaci nizu želiš: ";
      cin >> fibo;

      cout << fibo << ". broj u Fibonnaci nizu je: " << fibonacci(fibo) <<
endl;

      return 0;
}
```

Primjer koda izračunava N-ti broj u Fibonačijevom nizu (eng. Fibonacci sequence). Fibonačijev niz je niz brojeva koji se dobije tako da je sljedeći broj zbroj prethodna dva broja, dok je prvi broj u nizu nula, a drugi broj u nizu je jedan.

Kod radi tako da se unosi broj N koji predstavlja N-ti broj u nizu te se pomoću rekurzije taj broj izračuna. Rekurzija ima bazne uvjete, ukoliko je N nula vraća se broj nula, ukoliko je N jedan vraća se broj jedan. U ostalim slučajevima funkcija natrag poziva samu sebe ali uz druge parametre, u ovom slučaju N-1 te N-2 sve dok se ne dođe do baznog uvjeta.

# 4. Preopterećenje (overload)

Preopterećenje funkcija i metoda je sposobnost stvaranja više funkcija i metoda sa istim nazivom ali različitim implementacijama. Pozivi preopterećene funkcije će pokrenuti određenu implementaciju te funkcije koja odgovara kontekstu poziva. To omogućava da jedna funkcija izvršava različite zadatke ovisno o tome koja je potrebna.

Pravila za preopterećenje funkcija

1. Isto ime funkcije se koristi za više od jedne definicije funkcije
2. Funkcije moraju primati ili različiti broj parametara ili drugačiji tip

Primjer programskog koda za preopterećenje funkcija:

```
#include <iostream>
#include <cmath>
using namespace std;

int getPosInt();
int povrsina(int a); //varijanta 1
int povrsina(int a, int b); //varijanta 2
double povrsina(double r);

int main(void){
	int stranica_a = getPosInt();
	int stranica_b = getPosInt();
	double radijus = getPosInt();

	int povrsina_kvadra = povrsina(stranica_a);
	int povrsina_pravokutnika = povrsina(stranica_a, stranica_b);
	double povrsina_kruga = povrsina(radijus);

	cout << "Površina kvadra stranice " << stranica_a << " je " << povrsina_kvadra << "." << endl;
	cout << "Površina pravokutnika stranice " << stranica_a << " i stranice " << stranica_b << " je " << povrsina_pravokutnika << "." << endl;
	cout << "Površina kruga radijusa " << radijus << " je " << povrsina_kruga << "." << endl;

	return 0;
}
```

```
int getPosInt(){
	int broj = 1;

	do{
		cout << "Unesi pozitivni broj: ";
		cin >> broj;
	}while(broj <= 0);

	return broj;
}

int povrsina(int a){
	return pow(a,2);
}

int povrsina(int a, int b){ //overload po broju argumenta
	return (a * b);
}
double povrsina(double r){ //overload po tipu argumenta
	return r * 3.14;
}
```

## 4.1. Podudaranje argumenata

Preopterećene funkcije su odabrane za najbolje podudaranje deklaracija funkcija u trenutnom opsegu na argumente isporučene u funkcijskom pozivu. Ukoliko se pronađe odgovarajuća funkcija, ta funkcija se pozove. Upravo to se događa u primjeru. Funkcija *povrsina* je preopterećena te kod poziva se provjerava broj parametara i njihov tip podatka. Kada se pronađe odgovarajuće preopterećenje funkcije, ta funkcija će se pozvati. Pošto prvo i treće preopterećenje imaju jedan parametar, provjerava se tip podataka te se poziva preopterećenje funkcija sa odgovarajućim brojem argumenata i tipom podatka (u ovom slučaju prvi prvom pozivu se poziva varijanta 1).

## 4.2. Nadjačavanje

Nadjačavanje metode je pojam u objektno orijentiranom programiranju koje dozvoljava podklasi ili klasi djetetu da pruža određenu implementaciju metode koja je već definirana u klasi roditelja. Implementacija podklase nadjačava (zamjenjuje) implementaciju roditelja pružajući metodu koja ima isti naziv, parametre i povratni tip podatka kao i metoda

klase roditelja. Objekt koji se koristi određuje koja metoda će se izvršiti što omogućuje izvršavanje i metode roditelja.

Primjer nadjačavanja u c++:

```
#include <iostream>
using namespace std;

class cRoditelj{
	public:
		void poruka(){
			cout << "Ispis poruke roditelja." << endl;
		}
};

class cDijete : public cRoditelj{
	public:
		void poruka(){
			cout << "Ispis poruke dijeteta." << endl;
		}
};

int main(void){
	cRoditelj roditelj;
	cDijete dijete;

	roditelj.poruka();
	dijete.poruka();

	return 0;
}
```

U primjeru su dvije klase, cRoditelj i cDijete. Klasa cDijete nasljeđuje klasu cRoditelj te sve njene metode, ali metoda *poruka* klase cDijete nadjačava metodu klase cRoditelj stoga pri pozivu metode cDijete se ispisuje poruka klase dijeteta a ne klase roditelja.

Primjer nadjačavanja u Pythonu:

```
class cRoditelj(object):
    def __init__(self):
        pass
    def poruka(self):
        print ("Ispis poruke roditelja.")
```

```
class cDijete(cRoditelj):
    def __init__(self):
        super(cDijete, self).__init__()
    def poruka(self):
        print ("Ispis poruke dijete.")
        super(cDijete, self).poruka();

roditelj = cRoditelj()
roditelj.poruka()

dijete = cDijete();
dijete.poruka();
```

Kao i u prethodnom primjeru, imamo dvije klase, cRoditelj i cDijete gdje cDijete nasljeđuje klasu cRoditelj i njene metode. Prilikom ispisa poruke metoda klase cDijete nadjačava metodu klase cRoditelj. Python ujedno pruža i poziv metode superklase (odnosno roditelja) pomoću metode *super*.

# 5. Inline funkcije

Inline funkcije se koriste kako bi se smanjilo vrijeme izvršavanja programa. Funkcije se mogu uputiti kako se be „ugradile“ tako da programski prevoditelj može zamijeniti definiciju funkcije gdje se te funkcije pozivaju. Prevoditelj zamijenjuje definiciju ugrađene funkcije tokom prevođenja umjesto da se funkcija definira tokom izvršavanja. Kada se ugrađena funkcija pozove umjesto poziva funkcije cijeli kod funkcije se zamijenjuje sa pozivom što je vrlo značajno za vrijeme izvođenja manjih funkcija pošto je potrebno vrijeme za poziv funkcije dulje od vremena samo izvršavanja. „Ugrađivanje“ je samo zahtjev, a ne naredba prevoditelj te ju on može izostaviti. Sintaksa je sljedeća:

```
inline tip_funkcije ime_funkcije (parametri){
      tijelo funkcije
}
```

# 6. Indirektni poziv funkcije

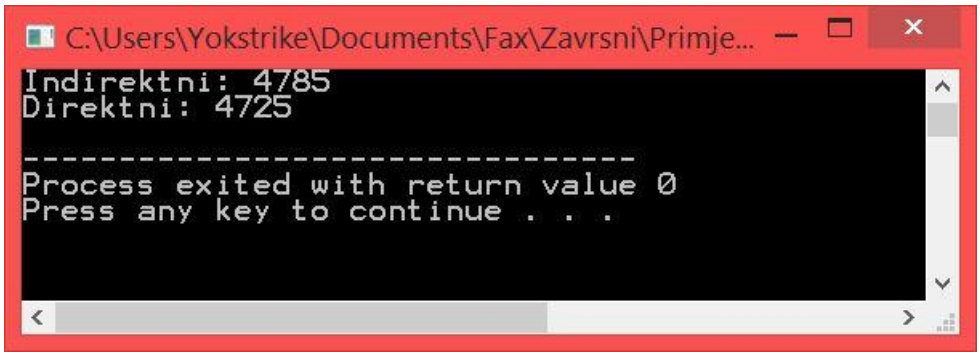
Kada c++ prevoditelj susretne definciju funkcije koja nije ugrađena (eng. inline) generira i pohranjuje strojni kod za tu funkciju u objektnoj datoteci te stvara naziv povezan sa strojnim kodom. U c++ to je tipično ime funkcije sa parametrima kako se bi se omogućila jedinstvena imena funkcija za svrhu preopterećivanja dok u C-u, koji ne podržava preopterećivanje, je to samo ime funkcije. Generalno prevoditelj ne zna adresu na kojoj je funkcija (recimo funkcija bi mogla biti u drugoj prevedenoj jedinici (prevedenu jedinicu stvori prevoditelj iz izvornog koda)), no ukoliko zna ime funkcije prevoditelj generira direktni poziv sa lažnom adresom. Nadalje, generira unos u objektnoj datoteci kako bi usmjerio povezivač na ažuriranu adresu funkcije sa tim imenom. Budući da povezivač vidi sve objektne datoteka stvorene iz svih jedinica, zna sva mjesta pozivanja kao i sva mjesta definiranja funkcija te zbog toga može popraviti sva mjesta poziva.

Primjer direktnog i indirektnog poziva funkcija:

```
#include <iostream>
#include <limits.h>
#include "biblioteka_vrijeme.cc"
using namespace std;

int f1(int a) {
   return a;
}

int direktni() {
   int i, b = 0;
   for (i = 0; i < INT_MAX; ++i) {
       f1(b);
   }
   return b;
}

int indirektni(int (*fn)(int)) {
   int i, b = 0;
      for (i = 0; i < INT_MAX; ++i){
            fn(b);
      }
   return b;
}
```

```
int main(void) {
	cvrijeme vrijeme;

	vrijeme.pocetak();
	indirektni(&f1);
	vrijeme.kraj();
	cout << "Indirektni: " << vrijeme.proteklo() << " ms" << endl;

	vrijeme.pocetak();
	direktni();
	vrijeme.kraj();
	cout << "Direktni: " << vrijeme.proteklo() << " ms" << endl;

	return 0;
}
```

Program sadrži dvije funkcije, jednu poziva direktno a drugi indirektno preko adrese te se prate vremena izvršavanja tih funkcija. Svaka funkcija vrti petlju do maksimalnog inta kako bi se dodalo vrijeme za izvršenje funkcije. Rezultat je sljedeći:




Slika 2. Rezultat indirektnog i direktnog poziva funkcija u milisekundama (prema vlastitoj izradi)

Pošto indirektni poziv funkcije dodatno dohvaća adresu direktni poziv je brži no vidimo da je razlika vrlo mala. Većina vremena potrošenog je na sam poziv funkcija, stog te samog izvršavanja koda unutar funkcija.

# 7. Zaključak

Potprogrami su blokovi naredbi koji izvršavaju jedan zadatak neke veće cjeline. Izvrsni i spretni potprogrami uvelike smanjuju trošak razvoja i održavanja velikog programa. Za poziv potprograma se koristi struktura podataka zvana kontrolni stog koja omogućuje spremanje lokalnih varijabli, slanje parametara te povratak na prethodnu instrukciju prije poziva potprograma.

Svaki potprogram može pozvati i samog sebe. Takvi potprogrami se nazivaju rekurzije. Rekurzije rješavaju veći problem tako da ga razbiju na više manjih dijelova koji prati isti algoritam kojim ga svode na bazni slučaj. Rekurzije se ponašaju kao i sve ostale funkcije te koriste stog za spremanje lokalnih vrijednosti, parametara i povratne adrese.

Potprogrami se mogu i preopteretiti i nadjačati. Preopterećenje potprograma je sposobnost stvaranja više potprograma sa istim nazivom ali različitim implementacijama što omogućava da jedan potprogram izvršava različite zadatke ovisno o tome koji je potreban. Nadjačavanje potprograma, odnosno metode, je pojam u objektno orijentiranom programiranju koje dozvoljava podklasi ili klasi djetetu da pruža određenu implementaciju metode koja je već definirana u klasi roditelja.

Potprogrami se mogu pozivati direktno ili indirektno. Indirektni poziv uključuje „ručno“ dohvaćanje adrese dok direktni poziv prepušta stvari prevoditelju. Na malim potprogramima je indirektni pristup sporiji zbog dodatnog dohvaćanja adrese no razlika je neznatna pošto je većina vremena potrošena na sam poziv potprograma, stog te izvršavanje koda unutar potprograma.

# Popis literature

(15. 1 2010). Preuzeto 2018 iz Codeidol: http://codeidol.com/community/cpp/direct-indirect-and-inline-calls/21809/

*Quora*. (2016). Preuzeto 2018 iz Quora: https://www.quora.com/What-is-a-call-stack-Is-there-a-good-resource-to-get-a-thorough-understanding-of-this-subject

Adamchik, V. S. (2009). Preuzeto 2018 iz Carnegie Mellon University: https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html

Baumgartner, A. (15. 12 2013). Preuzeto 2018 iz http://www.mathos.unios.hr/spa/Files/materijali/SPA_skripta_ch05.pdf

*GeeksForGeeks*. (n.d.). Preuzeto 2018 iz GeeksForGeeks: https://www.geeksforgeeks.org

Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language.* New Jersey: PRENTICE HALL.

Microsoft. (2018). Preuzeto 2018 iz https://msdn.microsoft.com/en-us/library/c4d5ssht.aspx

Oracle. (1997). *Brookhaven National Laboratory*. Preuzeto 2018 iz https://www.bnl.gov: https://www.bnl.gov/phobos/detectors/computing/orant/doc/database.804/a58236/07_subs.htm

Pigeon, S. (Prosinac 2008). Preuzeto 2018 iz https://hbfs.wordpress.com/2008/12/30/the-true-cost-of-calls/

# Popis slika

# Popis tablica